Homomorphic Encryption in MLOps using CKKS Approximate Arithmetics

Implementing Encrypted Stochastic Gradient Descent on Logistic Regression

Rayhan Kinan Muhannad – 13520065
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): rayhankinan@gmail.com

***Abstract*—In the recent years, the issue regarding security and data privacy in a machine learning system has been more prevalent than ever. One solution in developing a secure machine learning system is by implementing a homomorphic encryption into the training and deploying operations of a machine learning model. The author decided to use the CKKS (Cheong-Kim-Kim-Song) fully homomorphic encryption scheme to secure clients' sensitive data in a logistic regression model. The resulting F1 score metrics for model that trained on plaintext data and for the model that trained on ciphertext data shows no difference in term of magnitude. Although, there were some computational overheads in the form of time and resources needed for the training model by using ciphertext data. The overhead cost is significant enough that the industrial wide scale of the CKKS encrypted machine learning systems is yet possible given the scale and volume of data operated.**

Keywords—homomorphic encryption; CKKS encryption; logistic regression

I. Introduction

Data security is a crucial part of machine learning development to protect sensitive information, preserve privacy, maintain the integrity of models, and mitigate potential risks. Developing a machine learning model often requires access to large datasets that may contain sensitive information, such as personally identifiable information (PII), medical records, financial data, or proprietary business information. If these datasets are not adequately protected, there is a risk of exposing sensitive data to unauthorized individuals or entities. These machine-learning systems typically store and process large amounts of data, making them attractive targets for cyberattacks. A data breach can lead to unauthorized access, theft, or manipulation of sensitive data. It can have severe consequences, including financial loss, reputational damage, and violation of privacy regulations.

Machine learning models also can inadvertently learn or infer sensitive information from the training data, even if the data itself is not directly exposed to the engineer. This is known as "unintended memorization" or "model inversion" attacks. Unintended memorization is a term in machine learning that refers to a phenomenon where a machine learning model unintentionally learns or memorizes specific details or patterns present in the training data that are not relevant to the task it is supposed to solve. This can occur even when the sensitive information or patterns are not explicitly exposed or labelled in the dataset. When a model unintentionally memorizes sensitive information and the model is exposed to public use, it can raise serious privacy concerns. The model may exhibit a potential privacy risk by revealing sensitive information about individuals or entities during inference or by being susceptible to attacks that exploit the memorized information.

Other potential risks in developing a machine learning model are known as "adversarial attacks". Adversarial attacks aim to manipulate or deceive machine learning models by injecting malicious data or perturbations into the input. These attacks can lead to incorrect predictions, potentially compromising the integrity and security of the system. Adversarial attacks highlight the vulnerabilities and limitations of machine learning models. They demonstrate that even state-of-the-art models can be easily fooled by carefully crafted inputs, leading to potentially severe consequences in real-world scenarios. Adversarial attacks are especially concerning when dealing with sensitive applications such as healthcare or finance.

There are some solutions proposed for solving all those problems related to machine learning development. One of the most promising solutions is the use of homomorphic encryption in training and deploying machine learning models.

Homomorphic encryption is a cryptographic technique that allows for performing computations on encrypted data without the need for decryption. In other words, it enables data to remain encrypted while still allowing some mathematical operations (like addition and multiplication) to be performed on the encrypted data, producing results that are encrypted as well. The encrypted data can be processed by a third party or a computation server without the need to access the original plaintext data. Homomorphic encryption can be applied to various machine learning algorithms and models, making it a versatile solution for privacy-preserving machine learning. It supports training, inference, and other computations on encrypted data.

However, homomorphic encryption techniques usually come with significant computational overhead compared to

traditional computations on plaintext data. The encryption and decryption operations, as well as the computations on encrypted data, can be computationally intensive and slower.

After weighing the advantages and drawbacks of the use of homomorphic encryption in a machine learning system, the author decides to try implementing a simple model using a homomorphic encrypted dataset. The author decides that the easiest machine learning model to be implemented is the logistic regression trained using a stochastic gradient descent algorithm.

II. Theoritical Base

A. Homomorphic Encryption

Homomorphic encryption is a cryptographic technique that allows computations to be performed directly on encrypted data without requiring decryption. It enables performing operations on encrypted data while maintaining the privacy and confidentiality of the underlying information. This property makes homomorphic encryption particularly valuable in scenarios where sensitive data needs to be processed or analyzed while preserving privacy.

The fundamental idea behind homomorphic encryption is to design encryption algorithms in such a way that they support mathematical operations on ciphertexts, which, when decrypted, yield the same result as if the operations were performed on the corresponding plaintexts. This property is achieved using mathematical structures and techniques that enable computations on encrypted data.

There are different types of homomorphic encryption schemes, including partially homomorphic encryption (PHE) and fully homomorphic encryption (FHE). PHE schemes support a limited set of mathematical operations, such as addition or multiplication, on encrypted data. FHE schemes, on the other hand, support a broader range of operations, including arbitrary computations involving additions, multiplications, and even more complex operations like comparisons and logical operations.

Homomorphic encryption commonly involves three main operations:

- Encryption

  The process of converting plaintext data into encrypted form using a specific encryption algorithm. The encryption algorithm takes the plaintext data and a cryptographic key as input and produces the corresponding ciphertext.

- Homomorphic Operations

  These operations are performed on the encrypted ciphertexts, enabling mathematical computations to be performed directly on the encrypted data. Depending on the type of homomorphic encryption scheme, different operations can be supported, such as addition, multiplication, or more advanced operations.

- Decryption

  The process of converting the encrypted ciphertext back into plaintext form. Decryption requires the use of a corresponding decryption key, which is kept secret and only known to authorized parties. The decryption operation ensures that the result of the computation on the encrypted data is revealed in plaintext form.

The main drawbacks of homomorphic encryption schemes are that typically the scheme come with certain performance overhead due to the complexity of the cryptographic operations involved. The computational cost of performing operations on encrypted data is generally higher than performing the same operations on plaintext data.

B. CKKS Encryption Scheme

CKKS (Cheon-Kim-Kim-Song) is a homomorphic encryption scheme that provides a fully homomorphic encryption (FHE) solution for approximate computations on encrypted data. It is specifically designed to handle computations involving real or complex numbers, making it suitable for applications that require computations on continuous data, such as machine learning and statistical analysis.

The CKKS homomorphic encryption scheme extends the traditional binary-based homomorphic encryption schemes, like the Binary Encrypted Integer (BEI) scheme, to support operations on encrypted real or complex numbers. It achieves this by leveraging the concept of approximate arithmetic, where the computations are performed with some degree of precision loss.

The key components of CKKS encryption scheme are as follows:

- Parameter Generation

  The CKKS scheme begins by selecting appropriate parameters. This includes choosing prime numbers and generating polynomial rings over these primes. The choice of parameters impacts the security, efficiency, and precision of the computations.

- Key Generation

  The algorithm generates encryption keys and decryption key. The encryption key is used to encrypt plaintext values, while the decryption key is required to reveal the plaintext result after computation. Key generation involves random number generation and mathematical operations.

- Encoding

  Before encryption begins, plaintext values are encoded as polynomials. Each plaintext value is represented by a polynomial with coefficients that encode the value. The encoding process ensures that the plaintext values can be manipulated using the mathematical operations supported by the CKKS scheme.

  Some of the key aspects at encoding a CKKS plaintext are:

  - Polynomial Ring

    CKKS operates in a polynomial ring over a set of prime numbers. The choice of the prime numbers and their characteristics impact the precision and security of the

encoding process. The polynomial ring determines the maximum degree of the polynomials used for encoding.

- Scaling Factor

  CKKS employs a scaling technique to handle computations with different scales. The scaling factor is a power of 2 that determines the precision and dynamic range of the encoded values. The scaling factor affects the precision of the encoded values and the maximum range they can represent.

- Coefficient Generation

  To encode a plaintext value, a polynomial is constructed with coefficients that represent the value. The coefficients are generated based on the desired precision and the scaling factor. Typically, the coefficients are chosen from the integer range that corresponds to the precision and scaling factor.

- Normalization

  After generating the polynomial coefficients, normalization is performed. Normalization ensures that the polynomial coefficients fall within the desired range defined by the scaling factor. This step is important to maintain the precision and avoid numerical instability during computations.

- Encryption

  The encoding step is followed by the encryption of the encoded polynomials. Encryption transforms the polynomials into ciphertexts while preserving the privacy of the underlying plaintext values. The encryption process incorporates randomness and mathematical operations to hide the information contained in the polynomials.

  Some of the key aspects at encrypting a CKKS polynomial are:

  - Main Encryption Operation

    To encrypt a plaintext value, the encoded polynomial is combined with randomness. Randomness, or noise, is added to the polynomial to protect the privacy of the original plaintext. The randomness prevents an attacker from extracting information from the ciphertext.

  - Modular Reduction and Relinearization

    After encryption, the resulting ciphertext polynomial may have coefficients outside the desired range. Modular reduction is performed to bring the polynomial coefficients back within the desired range. Relinearization is an optional step that reduces the size of the ciphertext, making computations more efficient.

  - Serialization

    The encrypted polynomial, along with the public key and other necessary metadata, is serialized into a ciphertext object. The ciphertext object can be transmitted to a remote server or stored securely in a database.

- Homomorphic Operations

  CKKS supports homomorphic operations on the encrypted ciphertexts. These operations include addition, multiplication, and scaling (multiplication by a plaintext constant). Homomorphic operations can be performed directly on the ciphertexts without decrypting them, allowing computations on encrypted data.

  The author will explain each of the operations in the section below:

  - Addition

    Addition is a straightforward operation in CKKS that can be performed on encrypted ciphertexts. To add two ciphertexts, the corresponding ciphertext polynomials are added coefficientwise. The addition operation is performed modulo a predefined encryption parameter, typically represented by a large prime number. The resulting ciphertext represents the sum of the plaintext values encrypted in the original ciphertexts.

  - Multiplication

    Multiplication in CKKS is a more complex operation that requires additional steps to preserve privacy. To multiply two ciphertexts, the corresponding ciphertext polynomials are multiplied coefficientwise. Since multiplication can cause the polynomial degree to exceed the desired range, additional steps are needed to handle the overflow.

    The step by step of multiplication using encrypted data are as follows. The first step is to execute homomorphic multiplication using the predefined encryption parameter. This operation results in a ciphertext representing the product of the plaintext values encrypted in the original ciphertexts. The next step is to rescale the resulting multiplication. After the multiplication, the ciphertext may accumulate noise and have coefficients outside the desired range. Rescaling is performed to reduce the noise and adjust the polynomial degree. It involves dividing the ciphertext polynomial by a rescaling factor, which effectively reduces the noise level. The last step is to relinearize the rescaled result. Relinearization is an optional step that reduces the size of the ciphertext after multiplication and rescaling. It involves re-encrypting a subset of the ciphertext polynomials to reduce the overall ciphertext size. Relinearization is beneficial in terms of computational efficiency for subsequent homomorphic operations.

  - Scaling

    Scaling is an operation that allows adjusting the precision and magnitude of the encrypted values. It involves multiplying the ciphertext polynomial by a scaling factor. The scaling factor determines the precision of the encrypted values, and it can be chosen based on the desired level of accuracy. Scaling is often necessary to handle computations involving large numbers or to improve the accuracy of the encrypted results.

- Noise Management

As homomorphic operations are performed, noise is introduced into the ciphertexts. Noise arises from approximation errors and randomness introduced during encryption and homomorphic operations. To maintain correctness and security, the noise needs to be managed by periodically “refreshing” the ciphertexts or applying noise reduction techniques.

- Decryption

  After the desired computations are completed, the encrypted result can be decrypted to obtain an approximate plaintext result. The decryption operation uses the decryption key to transform the ciphertexts back into the original encoded polynomials. The decryption process involves reversing the encryption operations while accounting for the precision loss and noise.

- Decoding

  Decoding is the reverse process of encoding, where the encrypted polynomial ciphertexts are transformed back into approximate plaintext values. The decoding step allows us to reveal the approximate plaintext results of the computations performed on the encrypted data. It's important to note that decoding in CKKS involves precision loss due to the inherent approximations in the homomorphic encryption scheme. The precision loss is primarily due to the limited precision of the polynomial coefficients and the accumulated noise during the homomorphic operations.

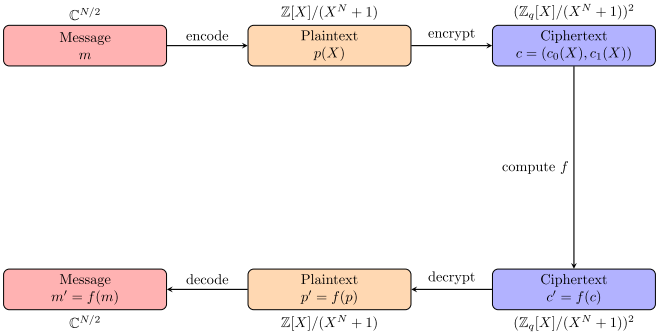

Fig. 1. CKKS Scheme for Encrypting and Decrypting

C. *Logistic Regression*

Logistic regression is a popular statistical and machine learning model used for binary classification tasks. It is a supervised learning algorithm that predicts the probability of an instance belonging to a particular class. In logistic regression, the goal is to model the relationship between a set of input features and a binary target variable. The target variable typically takes on two possible values, such as 0 and 1, or "negative" and "positive". The logistic regression model estimates the probability that a given instance belongs to the positive class based on the input features.

The key components in building a machine learning model using logistic regression are:

- Hypothesis Function (Sigmoid Function)

  The hypothesis function of logistic regression uses a logistic (or sigmoid) function to map the linear combination of input features to a value between 0 and 1.

$$\textit{Sigmoid Function}: \quad g(z) = \frac{1}{1+e^{(-z)}}$$

$$\textit{Hypothesis}: \quad h_\theta(x) = \frac{1}{1+e^{(-\theta^T x)}}$$

Fig. 2. Hypothesis of Logistic Regression Model

- Cost Function (Log Loss Function)

  The cost function used in logistic regression is the log loss (binary cross-entropy) function. The log loss measures the error between the predicted probabilities and the true labels. See in the figure below. If y = 1, then the cost = 0, and when the prediction = 0, the learning algorithm is punished by a very large cost. Similarly, if y = 0, predicting 0 has no punishment but predicting 1 has a large value of cost.

$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y = 1 \\ -log(1-h_\theta(x)) & \text{if } y = 0 \end{cases}$$

Fig. X. Cost Function of Logistic Regression Model

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)})$$

$$J(\theta) = \frac{1}{m}[\sum_{i=1}^{m} -y^{(i)} log(h_\theta(x^{(i)})) + (1-y^{(i)}) log(1-h_\theta(x^{(i)}))]$$

$$m = \textit{number of samples}$$

Fig. 3. Summation of Cost Function in Training Data

- Regularization in Cost Function (L1 and L2)

  Regularization is a technique used in logistic regression to prevent overfitting and improve the generalization performance of the model. It involves adding a regularization term to the cost function, which encourages the model to have smaller weights and reduces the complexity of the model. Two commonly used regularization techniques are L1 regularization (Lasso) and L2 regularization (Ridge).

  L1 regularization adds the sum of the absolute values of the weights to the cost function. L1 regularization has the effect of shrinking some of the weights towards zero, effectively performing feature selection. It can drive some weights to exactly zero, resulting in a sparse model.

  L2 regularization adds the sum of the squared values of the weights to the cost function. L2 regularization encourages

smaller weights but does not drive them to exactly zero. It penalizes large weights more strongly than L1 regularization.

Regularization helps to reduce overfitting by preventing the model from fitting the noise in the training data. It can improve the model's ability to generalize to unseen data by reducing the variance in the parameter estimates. However, it is important to strike a balance between regularization and model complexity, as excessive regularization can lead to underfitting.

$$L1: \quad J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \quad s.t. \quad \|\theta\|_1 <= C$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) + \frac{\lambda}{m}\sum_{j=1}^{n} |\theta_j|$$

$$L2: \quad J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \quad s.t. \quad \|\theta\|_2^2 <= C^2$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

$$m = number\ of\ samples, \quad n = number\ of\ features$$

Fig. 4. The Regularized Version of Log Loss Function

- Model Training (Stochastic Gradient Descent)

  Stochastic Gradient Descent (SGD) is an optimization algorithm commonly used in logistic regression (and other machine learning models) to find the optimal values of the model's parameters. It is a variant of the Gradient Descent algorithm that offers computational efficiency and scalability, especially for large datasets.

  There are three main steps in optimizing a logistic regression model using stochastic gradient descent. The first step is the initialization, where the model's parameters (weights and biases) are initialized with small random values. The next step is to iteratively optimize the parameters (weights and biases), in which for each mini-batch, the following steps are performed: Forward Propagation (the mini-batch is passed through the logistic regression model to compute the predicted probabilities for each example); Loss Calculation (the loss is computed based on the predicted probabilities and the true labels of the examples in the mini-batch); Backpropagation (the gradients of the loss with respect to the model's parameters are computed); and Parameter Update (the model's parameters (weights) are updated using the gradients and the learning rate). These steps are repeated for a specified number of iterations (epochs) or until convergence.

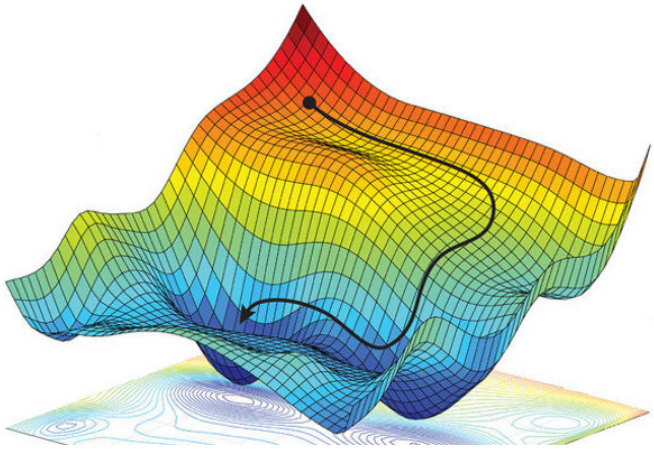

Fig. 5. The 3D Plot for Visualizing Stochastic Gradient Descent

- Model Evaluation

  Once the logistic regression model is trained, it can be used to make predictions on new, unseen beforehand data. Predicted probabilities can be calculated using threshold to make binary class predictions, and model performance can be evaluated using metrics such as accuracy, precision, recall, F1 score, or area under the ROC curve. There are some advantages and drawbacks in selecting one of those metrics to evaluate the model. Usually, the accuracy metrics are worse at describing a model behavior in less-than-ideal environment, as in an imbalanced dataset. Although the accuracy metrics does explain the model better in an ideal environment.

D. Encrypted Data Processing

Encrypted data processing using homomorphic encryption is a technique that allows performing computations on encrypted data without revealing the underlying plaintext values. It ensures the privacy and confidentiality of sensitive data while still enabling useful computations and analysis.

Homomorphic encryption schemes are designed to support mathematical operations on encrypted data. They allow data to be encrypted in a way that preserves its mathematical properties, enabling computations to be performed directly on the encrypted data. There are different types of homomorphic encryption, such as partially homomorphic encryption and fully homomorphic encryption (FHE).

The steps involved in encrypted data processing using homomorphic encryption are:

- Data Encryption

  The sensitive data is encrypted using an appropriate homomorphic encryption scheme. Each data point or attribute is transformed into an encrypted representation, called a ciphertext, using encryption algorithms. The encryption process ensures that the ciphertext maintains the mathematical structure of the plaintext data while obscuring its actual values.

- Homomorphic Operations

Homomorphic encryption schemes are designed to support specific mathematical operations, such as addition and multiplication, on the encrypted data. Depending on the type of homomorphic encryption used, it may support either addition or multiplication (partially homomorphic encryption) or both (fully homomorphic encryption). The encrypted data can undergo these supported operations directly without decryption, preserving the privacy of the underlying plaintext values.

- Computation on Encrypted Data

  Once the data is encrypted, computations can be performed on the ciphertexts using the supported homomorphic operations. For example, if the encrypted data represents numerical values, addition and multiplication operations can be applied to the ciphertexts to obtain results that correspond to the desired computations. The computations are performed on the ciphertexts without revealing the plaintext values, ensuring privacy.

- Result Decryption

  After the desired computations are completed on the encrypted data, the result is obtained as an encrypted ciphertext. To reveal the plaintext result, a decryption operation is performed on the encrypted ciphertext using a decryption key. The decryption process transforms the encrypted result back into the original plaintext form, allowing the computed result to be revealed.

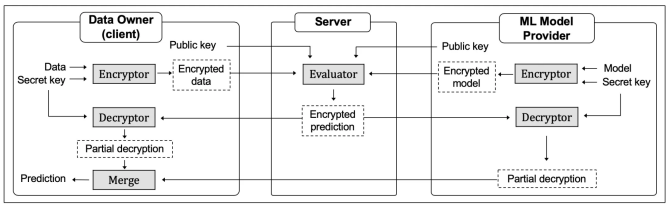

Fig. 6. A Secure Scheme of MLOps System using Homomorphic Encryption

III. IMPLEMENTATION

A. *Libraries and Frameworks*

To implement this program, the author applied and used many libraries and frameworks in the Python programming language. One of the reasons that the author employs many libraries and frameworks is that the program requires many of the tools needed for developing the machine learning model. Some of those libraries and frameworks are Pandas, Scikit-Learn, Imbalanced-Learn, Seaborn, Matplotlib, PyTorch, and TenSEAL. These libraries and frameworks play a crucial role in their respective field, like data cleaning, data visualization, building deep learning models, and encryption. We will be discussing each of those libraries in much greater detail in the next section.

The first library that the author used in this program is Pandas. Pandas is a library that eases the reading of CSV datasets, wrapping those datasets in a DataFrame object, and manipulating the content of the object by using methods and queries. By using Pandas, the author can freely read a CSV file and loads them into the program. The author also used Pandas for cleaning the dataset of null values or duplicate values which can cause bias in the model.

The author used Scikit-Learn and Imbalanced-Learn in tandem for complimenting the functionality of the Pandas library. Scikit-Learn is a complete library that can be used to instantiate basic machine learning models, split the main dataset into a train dataset and test dataset, and further clean the dataset of any impure data that can cause bias in the model. Mainly, the author uses Scikit-Learn to further clean the dataset of any bad data left and to split the dataset into datasets that are used for training only and dataset that are used for testing only. On the other hand, the author also used the Imbalanced-Learn library. Imbalanced-Learn is especially handy for handling imbalanced dataset that has a different composition of the number of the target class.

Furthermore, the author also used Matplotlib and Seaborn libraries to visualize some data that needed cleaning. At certain times, some datasets have an attribute that doesn't play that big of an effect on inferencing a target class or attributes that are highly dependent on other nontarget attributes. In developing a model, those attributes are sometimes omitted or conjoined with other, more important, attributes to minimize the dimension that the model operates on. The author can infer those attributes by using the heatmap plot of the dataset that is created by Matplotlib and Seaborn libraries.

The next library that the author used is PyTorch. PyTorch is mainly used for building more complex deep learning neural network models that require hardware integration like GPU to shorten the training time. The author extensively used the PyTorch library for building the logistic regression model to maximize the use of the GPU available on Google Colab. The basic data structure of PyTorch is the tensor, which PyTorch implements extensively in the neural network object. The tensor provides an easy abstraction for the program to execute on different types of machines, like multithreading on the CPU or thread block on the GPU. Not only that, but the PyTorch library also provides an extensive library for learning and calculating metrics of the models that the author trained.

Lastly, the library that the author used in this program is the TenSEAL. TenSEAL is a wrapper library in which its core implementation is the Microsoft SEAL library, written in C++. The author decided to use this library because the original CKKS encryption algorithm is first implemented by the research team at Microsoft using the SEAL library as is main tools. TenSEAL also integrate nicely with PyTorch because the backbone data structure of the library is the tensor, in which this also true on PyTorch. Although this is very convenience, the TenSEAL library is still relatively new and many of the optimization that the PyTorch has isn't yet implemented in TenSEAL. One of the most important optimizations that hasn't been implemented is the interface to GPU programming. This causes the executing time of the tensor operation to become significantly slower than those in PyTorch. Although, in overall functionality, the TenSEAL library provides more than enough for the author to develop a CKKS-based learning on logistic regression.

B. *Dataset Preparation*

One of the main cores of building a machine learning model is to find a suitable dataset that the model can efficiently predicts. The author spent a considerable time seeking and searching a dataset that are can be easily predicted by a logistic regression model. After researching, the author concludes that the dataset of an ongoing cardiovascular study on the residents of the town of Framingham, Massachusetts. The dataset contains all the necessary patients' health condition and whether they have a 10-year risk of a future heart coronary disease (CHD). The CSV file of this dataset is available on this URL: https://www.kaggle.com/datasets/dileep070/heart-disease-prediction-using-logistic-regression?resource=download

The author chooses this dataset for a couple of reasons. The first reason is that this dataset is said to be linearly separable, where there is a hyperplane that can separate the target class. The next reason that the author chooses this dataset is because the number of attributes (dimension) of the dataset is quite low, in which there were only 15 variable attributes to choose from. Although there were some drawbacks in this dataset. One of those drawbacks is that many of the variable attributes aren't independent of one another. Many of those attributes have a high correlation coefficient if the author plots them to a heatmap.

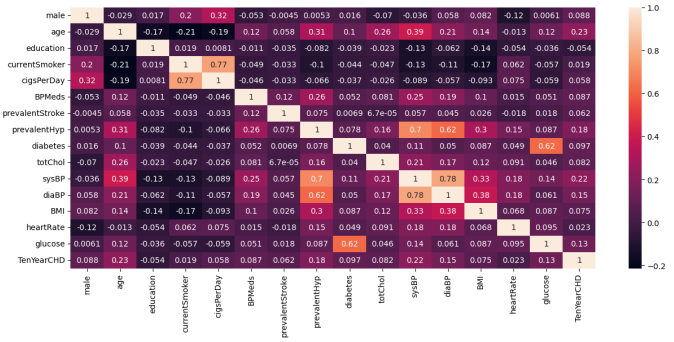

Fig. 7. Initial Heatmap of the Framingham Dataset

As you can see, the correlation coefficient between the attributes in the dataset is quite high, with some coefficients reaching 0.6 or even 0.7 in value. This can cause some complications to arise, like multicollinearity. Multicollinearity refers to a high degree of interdependence between the features. This can cause instability in the model and lead to unreliable coefficient estimates. Multicollinearity makes it challenging to interpret the impact of individual features on the target variable.

To solve that problem, the author decided to drop some of the attributes that are highly correlated to one another, like "prevalentHyp" that are highly correlated to "sysBP" and "diaBP"; "diabetes" with "glucose"; and "currentSmoker" with "cigsPerDay". The author also merged some attributes that are connected, like "sysBP" and "diaBP" which are merged into "meanArterialPressure" using a certain formula. Below is the resulting heatmap after dropping and merging those attributes.

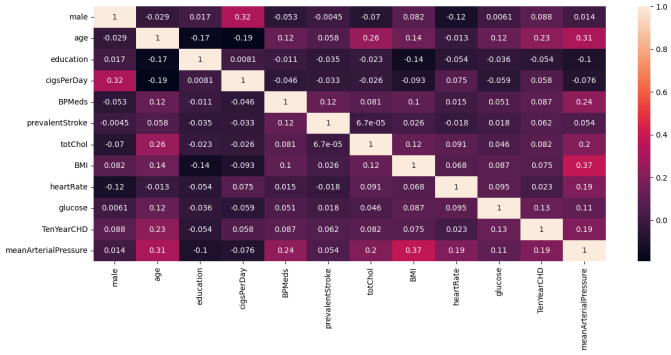

Fig 8. Resulting Heatmap of the Framingham Dataset

The next step after dropping correlated attributes is to clean the dataset of bad data, like null-valued data and duplicated data. This step is done to eliminate all noisy data that can cause bias in the resulting model.

After the resulting dataset is free from noise, the next step of the data preparation is to balance the previously imbalanced number of target classes in the dataset. One of the ways to correct this problem is to use sampling techniques to get an equivalent amount of the target class. There are many types of sampling, like under-sampling, over-sampling, and SMOTE. For this dataset, the author chooses the under-sampling technique so that the resulting dataset isn't too big (resource constraint).

Furthermore, after utilizing under-sampling of the dataset, the next step of the data preparation is to normalize the value of all numerical attributes. This is done to avoid biased weighting when the model is training. When attributes have different scales, the model may assign more importance or weight to attributes with larger values. Scaling attributes ensures that each attribute contributes proportionately to the learning process, avoiding biased weighting.

The last step of the data preparation is to split the resulting dataset into two parts, the training and testing dataset. The training dataset is used by the model to tune its weights and biases. The testing dataset is used to calculate the resulting metrics, like accuracy and F1 score. The author used a 0.2 ratio between the size of the training and testing dataset.

C. *Training on Regular Data*

After the dataset is prepared, the next step is to build the model itself. To compare the learning process on both the regular and the encrypted data, the author made two distinct model that share a common architecture. The author used logistic regression as its models' architecture. There are several reasons why the author chose logistic regression. The first reason is for simplicity. Logistic regressions are quite easy to build using tensor operation when compared with other models, like decision tree, SVM, or multilayered neural network. The second reason is that the sigmoid activation function used by logistic regression can be approximated in a certain range accurately using Remez algorithm. This property came in handy when developing a model in a constrained field, like in a CKKS ring field.

Initially, the author implemented a logistic regression class using a layer imported from PyTorch neural network library. Here, the author assumed that a logistic regression model is the same as a neural network model with only a single output layer and activated using a sigmoid function. The author also initialized the use of CUDA in tensor operation by adding it to the logistic regression model. The integration with CUDA would significantly improve the training and testing time of the model.

The next step is to transform the Pandas DataFrame object of the dataset into a PyTorch tensor object that are integrated to CUDA. This would be the basis of the dataset used for future PyTorch operation.

After the dataset is transformed into a tensor, the next step is to define the type of loss function and optimizer function used to tune the weight and bias tensor in the model. There were many loss-function and optimizer-function defined by the PyTorch library, all with their advantages and drawbacks. The author chose the "binary cross entropy" for the models' loss function and the "stochastic gradient descent" for the models' optimizer function. The reason that the author chose the two functions is for the ease to replicate the behavior of the two functions when implemented in encrypted data.

The last step to define before training the model is to define the metrics that would be used for calculating the "correctness" of the model. The author chose to use the metric "F1 score" rather than "accuracy" because of the imbalanced dataset, in which the F1 score metric would fare better than the accuracy metric.

After all parameters had been defined, the model can finally be trained using the combination of the loss-function "stochastic gradient descent" and optimizer-function "binary cross entropy" to calculate the change of weight and bias tensor in the layer using the train dataset. After the model had been trained, the model had to be evaluated using the "F1 score" metric to quantize the correctness of the model compared to the test dataset. Below are the resulting weights and biases of the trained logistic regression model.

```
Weight: [
0.2106858491897583,
0.27098724246025085,
-0.17342333495616913,
0.2575427293777466,
-0.02136250212788582,
0.06753615289926529,
-0.19143564999103546,
0.1304694563150406,
0.1681479811668396,
-0.22633618116378784,
0.2356201410293579
]
Bias: [-0.2182490974664688]
```

D. Training on Encrypted Data

There were some key notable differences in training using CKKS encrypted data when compared to training using regular data. The author had to define several additional parameters and change some data structures for the model to accept encrypted training data.

The first step in preparing the model to process encrypted data is to define the CKKS encryption parameter. There were several parameters of CKKS encryption that needed to be defined. The first parameter is the degree of polynomial modulus (scaling factor), which defines the encoding precision for the binary representation of the number. The SEAL library states that the degree of polynomial modulus must be a power of two. The degree of polynomial modulus also directly affects the number of coefficients in plaintext polynomial, the size of the ciphertext elements, the computational performance of the scheme (bigger is worse), and the security level (bigger is better).

Another parameter that had to be defined is the coefficient modulus sizes, which is the scheme for list of binary sizes. Using this list, the SEAL library will generate a list of primes of those binary sizes, called the coefficient modulus. The SEAL library states that the prime numbers in the coefficient modulus must be at most 60 bits and must be congruent to 1 modulo 2 * degree of polynomial modulus. The coefficient modulus sizes also directly affect the size of ciphertext elements, the length of list that indicates the level of the scheme (the number of multiplications allowed at a single time), and the security level (bigger is worse).

The next step after all the encryption parameter was defined is to encrypt all the dataset previously used. Note that the encryption time taken in each dataset is quite long because the author didn't integrate the operation to the CUDA device.

After the encryption parameter is settled, the next problem we had to solve is how to replicate the sigmoid function in a CKKS ring field. In CKKS ring field, the operation that can be ran between two CKKS operands are only addition and multiplication, where the inverse reciprocal operation is not defined. One solution to solve this problem is to approximate the sigmoid function by some polynomial function that can approximate the sigmoid function well. After reading some paper, the author came into conclusion that the best way to approximate the sigmoid function is by implementing the Remez algorithm. Remez algorithm works by calculating the polynomial for the best approximation of a function by applying the minimax approximation algorithm family. According to this paper (https://eprint.iacr.org/2018/462.pdf) that the author cites, the best approximation of the sigmoid function in range between –5 and 5 is the formula below:

$$f(x) \approx -0.004x^3 + 0.197x + 0.5, x \in [-5, 5]$$

Fig. 9. Remez Approximation of the Sigmoid Function

For the approximation to works, the resulting multiplication operation between the input and the weight vector followed by addition operation with the bias vector must be in range between

–5 and 5. The author anticipates this by scaling all the numerical attribute of the dataset into the range of 0 and 1 to minimize the distribution that lies out of range.

After the sigmoid function approximation is defined, the next step is to define the encrypted logistic regression class. There were several differences between the regular and the encrypted logistic regression class. For instance, in the encrypted logistic regression class, there were two options for doing the forward operation, which is the forward operation for encrypted input and the regular input. These two forward operations are called at different stages of model learning. The encrypted forward operation was called when the model is at the training phase, in which the data used as input are encrypted (server operation). The regular forward operation was called when the model is at the testing phase, in which the data used as input aren't encrypted (client operation).

Analogous to the logistic regression with regular training data, after all the parameters had been defined, the model can be trained and tested using the combination of loss-function "stochastic gradient descent", optimizer-function "binary cross entropy", and metric "F1 score". Below are the resulting weights and biases of the trained logistic regression model.

```
Weight: [
0.1294832224549116,
0.14004516821634833,
-0.04975166847947307,
0.022459932431842158,
0.04711431104768693,
0.009268722886341618,
0.009393104941824857,
0.013122961747297213,
-0.015721474469442026,
0.02044465432915576,
0.047737325585756205
]
Bias: [-0.04412464284436392]
```

IV. Analysis

Referring to the Google Colab notebook that the author had written, there were some key notable differences between the result in model trained using regular data (mainly utilize PyTorch operation) and the model trained using encrypted data (mainly utilize TenSEAL operation).

When comparing the result between the model that was using the regular and the model that was using the encrypted data, there were some notable differences. The first minor difference that if of the value of the F1 Score of each testing metrics. There were some explanations about this phenomenon. The first explanation is that the possibility that because of the approximate arithmetic nature of the CKKS encryption, there will be several imprecisions that can directly affects the F1 score metrics, in which these imprecisions would grow over linearly over time due to the number of operations done over the encryption. The second explanation is that the use of Remez algorithm to approximate the sigmoid function. The function for approximating the sigmoid function has a defined range that only works well if and only if the input is included in that range. It is entirely possible that the resulting operation of multiplication with the weight vector and the subsequent addition with the bias has the potential to generate a result that is outside of the range in the Remez algorithm.

```
[ ]  # Get F1 Score from Model

     test(logistic_regression, tensor_X_test, tensor_y_test)

     F1 Score: 0.674330
     Time Taken: 0.002502 s
```

Fig. 10. F1 Score of the Model using Regular Data

```
[ ]  # Client test the model using unencrypted data or infer using a new kind of data
     test(encrypted_logistic_regression, tensor_X_test.to("cpu"), tensor_y_test.to("cpu"))

     F1 Score: 0.677632
     Time Taken: 0.038348 s
```

Fig. 11. F1 Score of the Model using Encrypted Data

Another notable difference is the training time it took between the model that was using regular data and the model that was using encrypted data. The model that was using regular data had a much faster training time, that only of less than a second per training epoch (there were some overhead costs in compiling the code into GPU). This was a clear difference if it's compared to the model that was using encrypted data, where the training time per epoch of that model reaches five minutes (averaging just below 300 seconds). The training time differences and the resource that it took is significant enough that not every server that has the capacity in training the model using regular data can handle training the model in encrypted data. Because of that overhead training time, the widespread implementation of CKKS encryption scheme in machine learning systems is hindered. The amount of time that it took to accomplish a single task using encrypted data is deal breaker for companies that is trying to secure their machine learning systems. There exists many more method for securing their systems trade off more security in favor of performance, like the federated learning techniques. Compared to homomorphic encryption, the federated learning has a system that are more vulnerable to attacks but a much faster and efficient in term of time and resources needed.

```
Epoch 1
-------------------------------
Loss: 0.699757
Time Taken: 3.040356 s

Epoch 2
-------------------------------
Loss: 0.685218
Time Taken: 0.001012 s

Epoch 3
-------------------------------
Loss: 0.680260
Time Taken: 0.001032 s

Epoch 4
-------------------------------
Loss: 0.677913
Time Taken: 0.000959 s

Epoch 5
-------------------------------
Loss: 0.676272
Time Taken: 0.000932 s
```

Fig. 12. Training Time of the Model using Regular Data

```
Epoch 1
-------------------------------
Time Taken: 290.044480 s

Epoch 2
-------------------------------
Time Taken: 289.288920 s

Epoch 3
-------------------------------
Time Taken: 289.419346 s

Epoch 4
-------------------------------
Time Taken: 288.835741 s

Epoch 5
-------------------------------
Time Taken: 287.516435 s
```

Fig. 13. Training Time of the Model using Encrypted Data

V. Conclusion

The need of creating a secure and privacy preserving machine learning systems is continuously rising in the last decade or so. One of the best solutions in developing a secure machine learning systems is to implement a homomorphic encryption into the training and deploying operations of a machine learning models. There are two types of homomorphic encryption: partially homomorphic encryption and fully homomorphic encryption. In a partially homomorphic encryption, only one of the two main operations in a ring field (addition and multiplication) are supported by the encrypted operation. In a fully homomorphic encryption, the two main operations of addition and multiplication are supported by the encrypted operation. Therefore, a fully homomorphic encryption is better option to use in a machine learning systems than a partially homomorphic encryption.

One of the examples of a fully homomorphic encryption algorithm is the CKKS scheme. The CKKS scheme utilize an approximate arithmetic in its internal calculation. The approximation arithmetic can sometime cause inaccuracy when calculating a fix point value. Fortunately, in a machine learning system, the calculation of the model mostly dealt with a floating-point dataset and can dealt with certain threshold of inaccuracies.

Using the CKKS scheme, the author can implement a MLOps system in which the model is hosted at a remote server. Then, the client could have sent their own sensitive data that were encrypted using CKKS over an unsecured connection to the server. Then, the server can train a certain model using the clients' encrypted data to tune the parameter of the models. The resulting parameters will be homomorphic to the parameters of the model that are trained using ordinary data. After the server trained the model, the client could have sent some new encrypted input to the model to be inferenced. The model would calculate the new encrypted output and sent it back to the clients. After that, the clients can decrypt the received output from server and use it in their local machine.

The main drawback of the use of CKKS scheme in a machine learning system is that the overhead cost of training using encrypted data is still quite high. The operations needed for a single addition or multiplication on an encrypted data is significantly higher than those done on regular data. To implement this in an industrial scale, the author will need to optimize the calculations of encrypted data. One of those optimizations needed is the use of hardware accelerations, like GPUs, to further reduce the training time of models.

Source Code Link at Google Colab

The work containing all the authors' implementations and programs is contained in this Google Colab notebook: https://colab.research.google.com/drive/1eA8xgFdiXPwhehq0NAnLtokspjjGp60d?usp=sharing

Acknowledgment

The author is grateful to Mr. Rinaldi Munir, the primary lecturer of the IF4020 Cryptography who taught and guide the author the subject of cryptography so much so that the author can finish this paper. The author also thankful to all of friends and family that supported the author with time and resources to complete this paper.

References

[1] Song et al. 2016. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. https://eprint.iacr.org/2016/421.pdf, accessed on May 15th, 2023.

[2] Benaissa, Ayoub. 2020. *Homomorphic Encryption in PySyft With SEAL and PyTorch*. https://blog.openmined.org/ckks-homomorphic-encryption-pytorch-pysyft-seal/, accessed on May 17th, 2023.

[3] Lopardo et al. 2020. *What is Homomorphic Encryption?*. https://blog.openmined.org/what-is-homomorphic-encryption/, accessed on May 17th, 2023.

[4] Huynh, Daniel. 2020. *CKKS Explained: Part 1, Vanilla Encoding and Decoding*. https://blog.openmined.org/ckks-explained-part-1-simple-encoding-and-decoding/, accessed on May 19th, 2023.

[5] Huynh, Daniel. 2020. *CKKS Explained: Part 2, Full Encoding and Decoding*. https://blog.openmined.org/ckks-explained-part-2-ckks-encoding-and-decoding/, accessed on May 19th, 2023.

[6] Huynh, Daniel. 2020. *CKKS Explained: Part 3, Encryption and Decryption*. https://blog.openmined.org/ckks-explained-part-3-encryption-and-decryption/, accessed on May 20th, 2023.

[7] Huynh, Daniel. 2020. *CKKS Explained: Part 4, Multiplication and Relinearization*. https://blog.openmined.org/ckks-explained-part-4-multiplication-and-relinearization/, accessed on May 20th, 2023.

[8] Huynh, Daniel. 2020. *CKKS Explained: Part 5, Rescaling*. https://blog.openmined.org/ckks-explained-part-5-rescaling/, accessed on May 20th, 2023.

[9] Chen et al. 2018. *Logistic Regression over Encrypted Data from Fully Homomorphic Encryption*. https://eprint.iacr.org/2018/462.pdf, accessed on May 21st, 2023.

STATEMENT

I hereby declare that this paper is of my own writing, not an adaptation or a translation of other people's papers, nor plagiarism.

Bandung, 22nd of May 2023